

CLASSES AND OBJECTS

MODULE 2



INTRODUCTION TO OOP

Content

- > Introduction to Software Engineering
- > Object-Oriented Programming Paradigm

Software

- > Computer Software is the product that software engineers design and build.
- > It encompasses
 - programs that execute within a computer of any size and architecture,
 - documents that encompass hard-copy and virtual forms,
 - data that combine numbers and text but also includes representations of pictorial, video and audio information.

Examining Object Orientation

OO concepts affect the whole development process:

- > Humans think in terms of nouns (objects) and verbs (behaviors of objects).
- > With OOSD, both problem and solution domains are modeled using OO concepts.
- > The *Unified Modeling Language* (UML) is a de facto standard for modeling OO software.
- > OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

Examining Object Orientation

“Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the objects types than on the actions.” (Meyer page vi)

OO concepts affect the following issues:

- > Software complexity
- > Software decomposition
- > Software costs

Software Complexity

- > The profile and importance of software has increased



Software Complexity is increasing

Product	Lines of code
Order entry system	1.7 million
F-22 Raptor	1.7 million
Space Shuttle	2 million
Microsoft Word	2 million (27K in 1 st release)
F-35 Joint Strike Fighter	5.7 million
Airline reservation system	6.1 million
Boeing 787 Dreamliner	6.5 million
S Class Mercedes-Benz Radio w/ Nav system	20 million

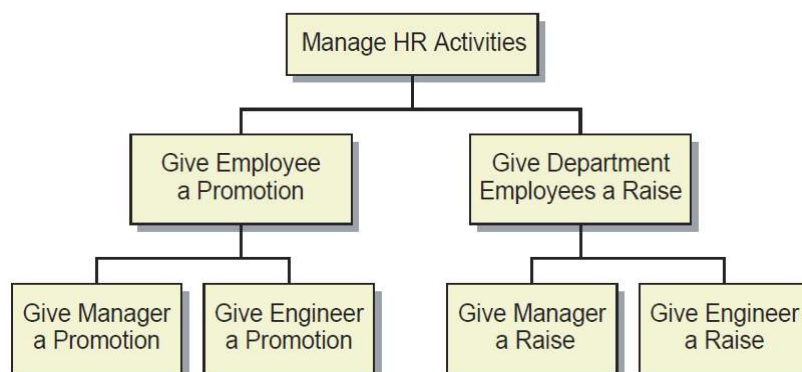
Software Complexity

Complex systems have the following characteristics:

- > They have a **hierarchical structure**.
- > The choice of **which components are primitive** in the system is arbitrary.
- > A system can be split by intra- and inter-component relationships. This **separation of concerns** enables you to study each part in relative isolation.
- > Complex systems are usually composed of only a **few types of components in various combinations**.
- > A successful, complex system invariably **evolves from a simple working system**.

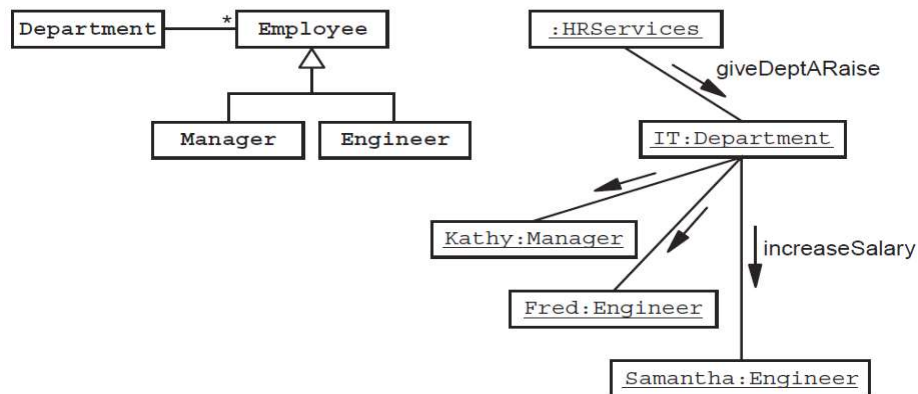
Software Decomposition

- > In the Procedural paradigm, software is decomposed into a hierarchy of procedures or tasks.



Software Decomposition

- > In the OO paradigm, software is decomposed into a hierarchy of interacting components (usually objects).



Software Costs

Development:

- > OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.
- > OO-modeled business entities and processes are easier to implement in an OO language.

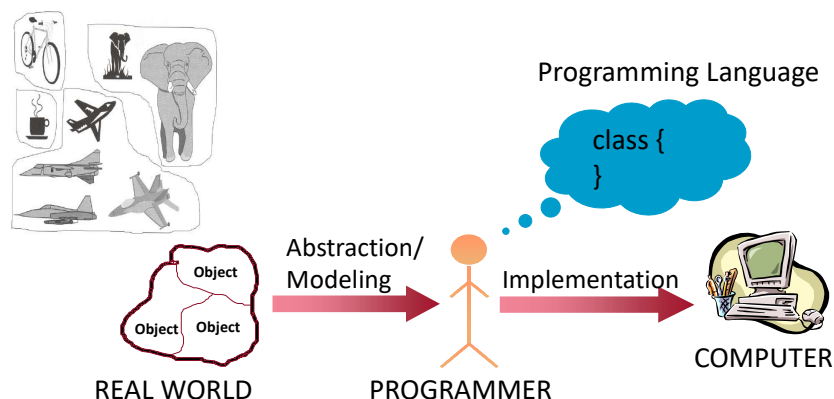
Maintenance:

- > Changeability, flexibility, and adaptability of software is important to keep software running for a long time.
- > OO-modeled business entities and processes can be adapted to new functional requirements.

Why C++

- > C++ supports writing high quality programs (supports OO)
- > C++ is used by hundreds of thousands of programmers in every application domain.
 - This use is supported by hundreds of libraries, hundreds of textbooks, several technical journals, many conferences.
- > Application domain:
 - Systems programming: Operating systems, device drivers. Here, direct manipulation of hardware under real-time constraints are important.
 - Banking, trading, insurance: Maintainability, ease of extension, ease of testing and reliability is important.
 - Graphics and user interface programs
 - Computer Communication Programs

What is Programming?



- > If successful, this medium of expression (the object-oriented way) will be significantly easier, more flexible, and efficient than the alternatives as problems grow larger and more complex.



Learning C++ (1/2)

- > Like human languages, programming languages also have *many syntax* and *grammar rules*.
- > Knowledge about grammar rules of a programming language is *not enough* to write “*good*” programs.
- > The most important thing to do when learning C++ is to *focus on concepts* and not get lost in language-technical details.
- > *Design techniques* is far *more important* than an understanding of details; that understanding comes with time and practice.
- > Before the rules of the programming language, the programming scheme must be understood.

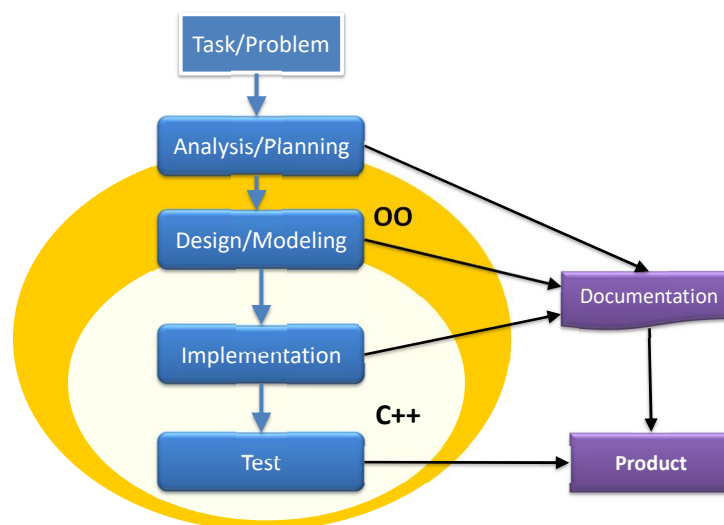
Learning C++ (2/2)

- > Your purpose in learning C++ must not be simply to learn a new syntax for doing things the way you used to, but to *learn new* and *better ways of building systems*

Software Quality Metrics

- A program must do its job correctly. It must be useful and usable.
 - A program must perform as fast as necessary (Real-time constraints).
 - A program must not waste system resources (processor time, memory, disk capacity, network capacity) too much.
 - It must be reliable.
 - It must be easy to update the program.
 - A good software must have sufficient documentation (users manual).
- 
- Source code must be readable and understandable.
 - It must be easy to maintain and update (change) the program.
 - A program must consist of independent modules, with limited interaction.
 - An error may not affect other parts of a program (Locality of errors).
 - Modules of the program must be reusable in further projects.
 - A software project must be finished before its deadline.
 - A good software must have sufficient documentation (about development).
- 

Software Development Process



Software Development Process

> Analysis

- Gaining a clear understanding of the problem. Understanding requirements. They may change during (or after) development of the system!
- Building the programming team.

Software Development Process

> Design

- Identifying the key concepts involved in a solution. Models of the key concepts are created.
- This stage has a strong effect on the quality of the software.
- Therefore, before the coding, verification of the created model must be done.
- Design process is connected with the programming scheme. Here, our design style is object-oriented.

Software Development Process

> Coding

- The solution (model) is expressed in a program.
- Coding is connected with the programming language.
- In this course we will use C++.

Software Development Process

> Documentation

- Each phase of a software project must be clearly explained.
- A users manual should be also written.

Software Development Process

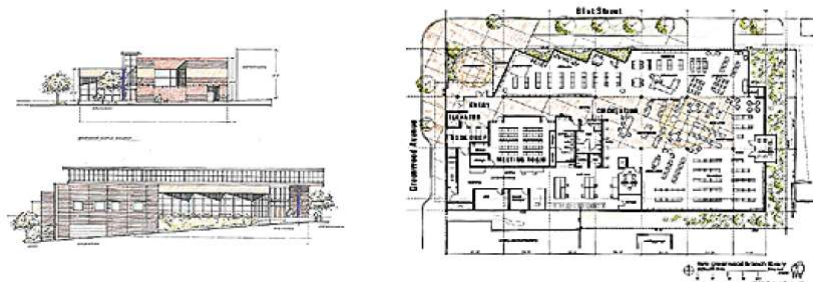
> Test

- The behavior of the program for possible inputs must be examined.

What is a Model?

“A model is a simplification of reality.”

(Booch UML User Guide page 6)



- > A model is an abstract conceptualization of some entity (such as a building) or a system (e.g. software).
- > Different views show the model from different perspectives.

Why Model Software?

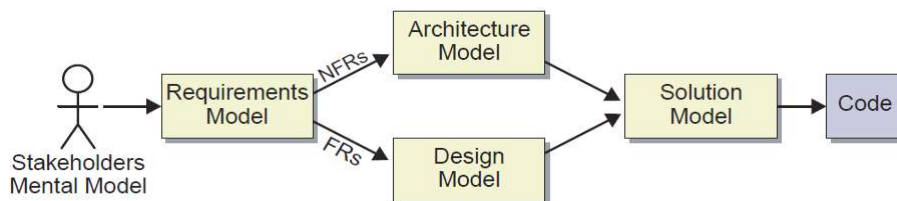
“We build models so that we can better understand the system we are developing.”

(Booch UML User Guide page 6)

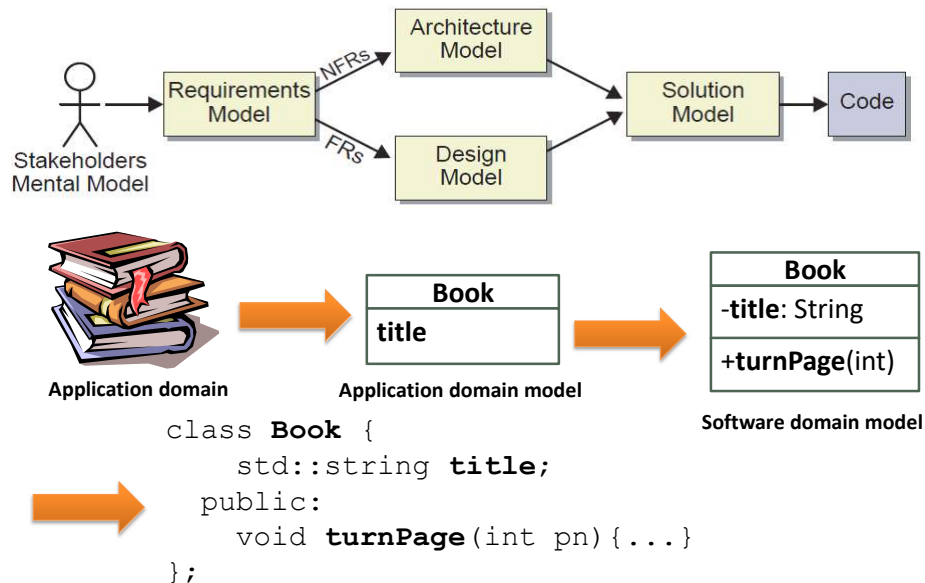
- > Specifically, modeling enables us to:
 - Visualize new or existing systems
 - Communicate decisions to the project stakeholders
 - Document the decisions made in each OOSD workflow
 - Specify the structure (static) and behavior (dynamic) elements of a system
 - Use a template for constructing the software solution

OOSD as Model Transformations

- > Software development can be viewed as a series of transformations from the Stakeholder’s mental model to the actual code:



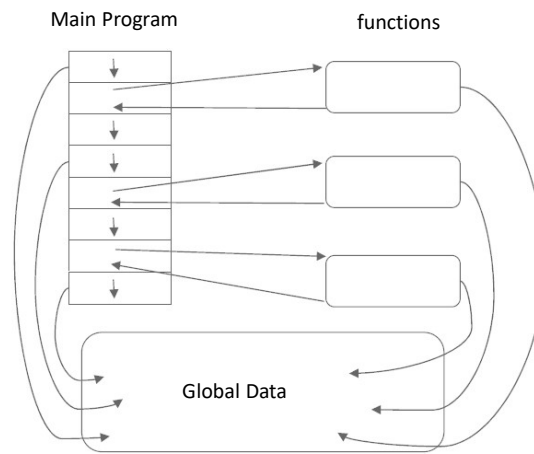
OOSD as Model Transformations



Procedural Programming

- > Pascal, C, BASIC, Fortran, and similar traditional programming languages are procedural languages. That is, each statement in the language tells the computer to do something.
- > In a procedural language, the emphasis is on doing things (functions).
- > A program is divided into functions and—ideally, at least—each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.

Procedural Programming



Problems with Procedural Programming

- > Data is **Undervalued**
- > Data is, after all, the reason for a program's existence.
- > Procedural programs (functions and data structures) don't model the real world very well.
 - The real world does not consist of functions.
 - Global data can be corrupted by functions that have no business changing it.
- > To add new data items, all the functions that access the data must be modified so that they can also access these new items.
- > Creating new data types is difficult.

Besides...

- > It is also possible to write good programs by using procedural programming (C programs).
- > But object-oriented programming offers programmers many advantages, to enable them to write high-quality programs.

Object Oriented Programming

- > The fundamental idea behind OOP is:
 - The real world consists of objects.
 - Real world objects have two parts:
 - Properties (or state :characteristics that can change)
 - Behavior (or abilities :things they can do).
 - To solve a programming problem in an object-oriented language, the programmer no longer asks how the problem will be divided into functions, but how it will be divided into objects.
 - The emphasis is on **data**

Object Oriented Programming

- > What kinds of things become objects in object-oriented programs?
 - Human entities: Employees, customers, salespeople, worker, manager
 - Graphics program: Point, line, square, circle, ...
 - Mathematics: Complex numbers, matrix
 - Computer user environment: Windows, menus, buttons
 - Data-storage constructs: Customized arrays, stacks, linked lists

Surveying the Fundamental OO Concepts

- > Objects
- > Classes
- > Abstraction
- > Encapsulation
- > Inheritance
- > Interfaces
- > Polymorphism
- > Cohesion

Objects

object = state + behavior

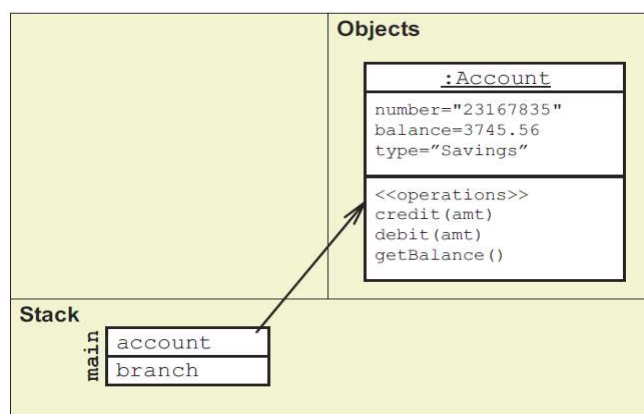
“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”

(Booch Object Solutions page 305)

Objects:

- > Have identity
- > Are an instance of only one class
- > Have attribute values that are unique to that object
- > Have methods that are common to the class

Objects: Example



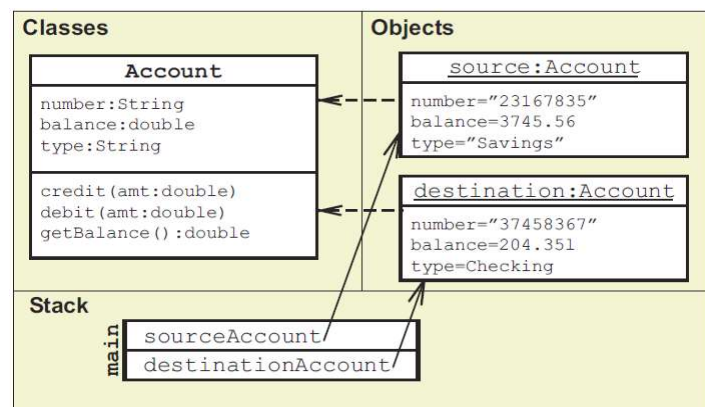
Classes

A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

Classes provide:

- > The metadata for attributes
- > The signature for methods
- > The implementation of the methods (usually)
- > The constructors to initialize attributes at creation time

Classes: Example



Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- > The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- > The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

Abstraction: Example

Engineer
fname:String lname:String salary:Money
increaseSalary(amt) designSoftware() implementCode()

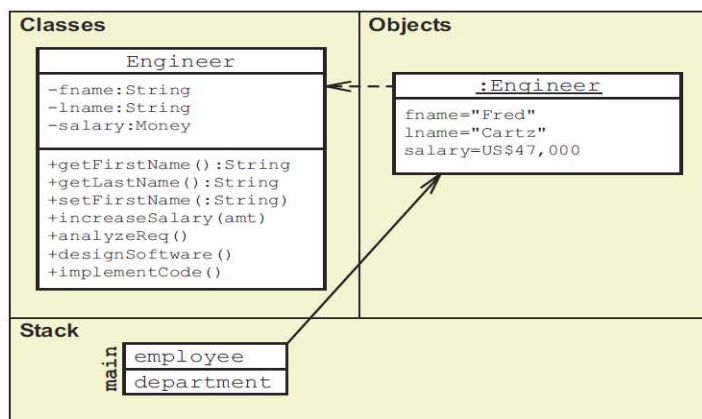
Engineer
fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

Encapsulation

Encapsulation means “to enclose in or as if in a capsule” (Webster New Collegiate Dictionary)

- > Encapsulation is essential to an object. An object is a capsule that holds the object’s internal state within its boundary.
- > In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods”.

Encapsulation: Example



✗ `name = employee.fname;`
✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`
✓ `employee.setFirstName("Samantha");`

Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

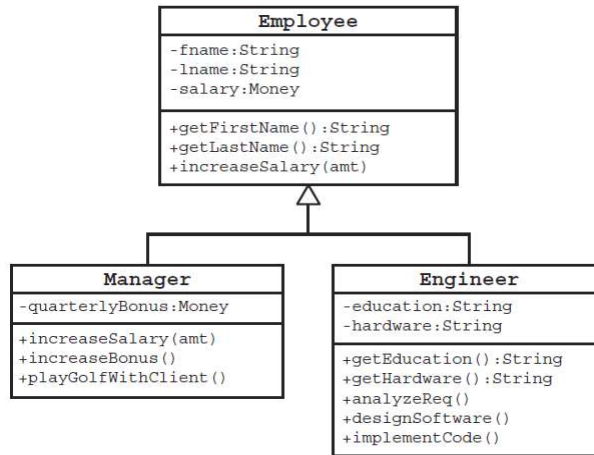
Features of inheritance:

- > Attributes and methods from the superclass are included in the subclass.
- > Subclass methods can override superclass methods.
- > The following conditions must be true for the inheritance relationship to be plausible:
 - A subclass object *is a (is a kind of)* the superclass object.
 - Inheritance should conform to Liskov’s Substitution Principle (LSP).

Inheritance

- > Specific OO languages allow either of the following:
 - Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).
 - Multiple inheritance, which allows a class to directly inherit from one or more superclasses (for example, C++).

Inheritance: Example



Abstract Classes

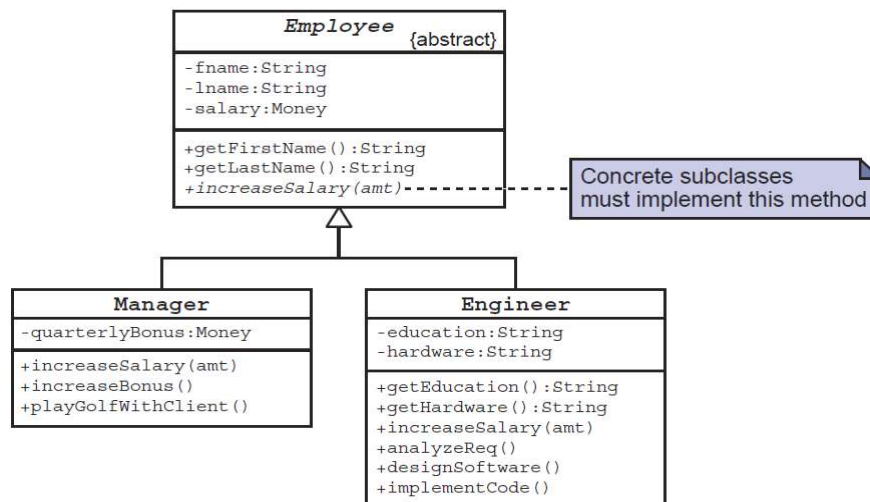
A class that contains one or more abstract methods, and therefore can never be instantiated.

Features of an abstract class:

- > Attributes are permitted.
- > Methods are permitted and some might be declared abstract.
- > Constructors are permitted, but no client may directly instantiate an abstract class.
- > Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.

Abstract Classes: Example

- > In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with **{abstract}**.



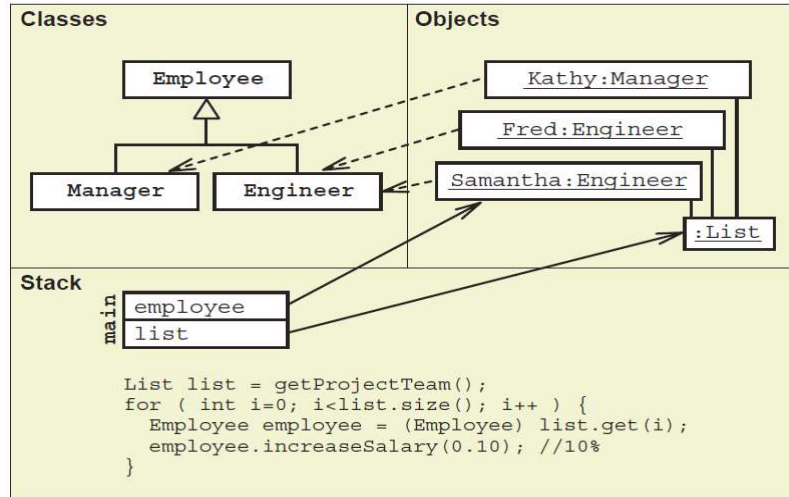
Polymorphism

Polymorphism is “a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type].” (Booch OOAD page 517)

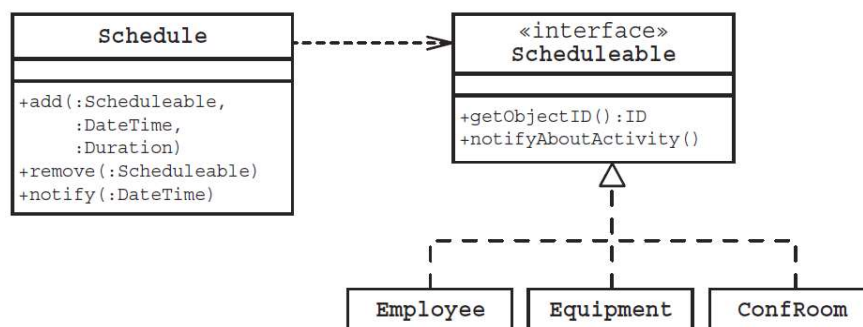
Aspects of polymorphism:

- > A variable can be assigned different types of objects at runtime provided they are a subtype of the variable’s type.
- > Method implementation is determined by the type of object, not the type of the declaration (dynamic binding).
- > Only method signatures defined by the variable type can be called without casting.

Polymorphism: Example



Polymorphism: Example



Cohesion

- > In software, cohesion refers to how well a given component or method supports a single purpose.
 - Low cohesion occurs when a component is responsible for many unrelated features.
 - High cohesion occurs when a component is responsible for only one set of related features.
 - A component includes one or more classes. Therefore, cohesion applies to a class, a subsystem, and a system.
 - Cohesion also applies to other aspects including methods and packages.
 - Components that do everything are often described with the Anti-Pattern term of Blob components.

Cohesion: Example

Low Cohesion

SystemServices
makeEmployee makeDepartment login logout deleteEmployee deleteDepartment retrieveEmpByName retrieveDeptByID

High Cohesion

LoginService
login logout
EmployeeService
makeEmployee deleteEmployee retrieveEmpByName
DepartmentService
makeDepartment deleteDepartment retrieveDeptByID

OOP vs. Procedural Programming

Procedural Programming

- > Procedural languages still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve.
- > The programmer must establish the association between the machine model and the model of the problem that is actually being solved.
- > The effort required to perform this mapping produces programs that are difficult to write and expensive to maintain. Because the real world thing and their models on the computer are quite different.

Example: Procedural Programming

- > Real world thing: Customer, Employee, Account
- > Computer model: char *, int, float ...
- > It is said that the C language is closer to the computer than the problem.

OOP vs. Procedural Programming

Object Oriented Programming

- > The object-oriented approach provides tools for the programmer to represent elements in the problem space.
- > We refer to the elements in the problem space and their representations in the solution space as “objects.”
- > The idea is that the program is allowed to adapt itself to the problem by adding new types of objects, so when you read the code describing the solution, you’re reading words that also express the problem.
- > OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run.

OOP vs. Procedural Programming

- > Benefits of the object-oriented programming:
 - Readability
 - Understandability
 - Low probability of errors
 - Maintenance
 - Reusability
 - Teamwork

CLASSES AND OBJECTS

Content

> OOP Concepts

– **Class**

- **Encapsulation**

- **Information Hiding**

– Inheritance

– Polymorphism



Black-Box Design

OOP Concepts

- > When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions, but **how it will be divided into objects**.
- > Thinking in terms of objects rather than functions has a helpful effect on how easily you can design programs. Because **the real world consists of objects** and there is a close match between objects in the programming sense and objects in the real world.

What is an Object?

- > Many real-world objects have both a **state** (characteristics that can change) and **abilities** (things they can do).
- > Real-world object=State (properties)+ Abilities (behavior)
- > Programming objects = Data + Functions
- > The match between programming objects and real-world objects is the result of combining data and member functions.
- > How can we define an object in a C++ program?

Classes and Objects

- > **Class** is a new data type which is used to model objects.
- > A class serves as a plan, or a template. It specifies what data and what functions will be included in objects of that class.
- > Writing a class doesn't create any objects.
- > A class is a description of similar objects.
- > **Objects** are instances of classes.

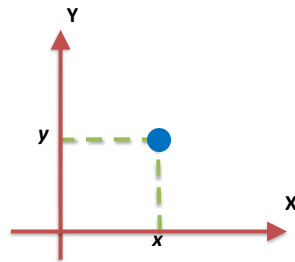
Example

A model (class) to define points in a graphics program.

- > Points on a plane must have two properties (states):
 - **x** and **y** coordinates. We can use two integer variables to represent these properties.
- > In our program, points should have the following abilities (behavior):
 - Points can move on the plane: **move** function
 - Points can show their coordinates on the screen: **print** function
 - Points can answer the question whether they are on the zero point (0,0) or not: **isZero** function

Class Definition: Point

```
class Point {                                // Declaration of Point Class
    int x,y;                                // Properties: x and y coordinates
public:                                     // We will discuss it later
    void move(int,int); // A function to move the points
    void print();        // to print the coordinates on the screen
    bool isZero();       // is the point on the zero point(0,0)
};                                           // End of class declaration (Don't forget ;)
```



Bodies of Member Functions

```
// A function to move the points
void Point::move(int new_x, int new_y) {
    x = new_x;    // assigns new value to x coordinate
    y = new_y;    // assigns new value to y coordinate
}

// To print the coordinates on the screen
void Point::print() {
    cout << "X= " << x << ", Y= " << y << endl;
}

// is the point on the zero point(0,0)
bool Point::isZero() {
    // if x=0 & y=0 returns true
    return (x == 0) && (y == 0);
}
```


Class Definition: Point

- > In our example first data and then the function prototypes are written.
- > It is also possible to write them in reverse order.
- > Data and functions in a class are called members of the class.
- > In our example only the prototypes of the functions are written in the class declaration. The bodies may take place in other parts (in other files) of the program.
- > If the body of a function is written in the class declaration, then this function is defined as an inline function (macro).

```
int main() {  
    Point point1, point2; // 2 object are defined: point1 and point2  
    point1.move(100,50); // point1 moves to (100,50)  
    point1.print();      // point1's coordinates to the screen  
    point1.move(20,65);   // point1 moves to (20,65)  
    point1.print();      // point1's coordinates to the screen  
    if( point1.isZero() ) // is point1 on (0,0)?  
        cout << "point1 is now on zero point(0,0)" << endl;  
    else  
        cout << "point1 is NOT on zero point(0,0)" << endl;  
    point2.move(0,0); // point2 moves to (0,0)  
    if( point2.isZero() ) // is point2 on (0,0)?  
        cout << "point2 is now on zero point(0,0)" << endl;  
    return 0;  
}
```

```

class Time {
    int hour;
    int minute;
    int second ;
public:
    // Getter Methods
    int  getHour(){return hour;};
    int  getMinute(){return minute;};
    int  getSecond (){return second;};
    // Setter Methods
    void setTime(int h,int m,int s){hour=h;minute=m;second=s;};
    void setHour(int h){hour= (h>=0 && h<24) ? h : 0;};
    void setMinute(int m){minute= (m>=0 && m<60) ? m : 0;};
    void setSecond(int s){second= (s>=0 && s<60) ? s : 0;};
};

```

UML Class Diagram

Time
-hour:short -minute:short -second:short
+getHour():short +getMinute():short +getSecond():short +setHour(hour:short) +setMinute(minute:short) +setSecond(second:short)

C++ Terminology

- > A **class** is a grouping of data and functions. A class is very much like a structure type as used in ANSI-C, it is only a pattern to be used to create a variable which can be manipulated in a program.
- > An **object** is an instance of a class, which is similar to a variable defined as an instance of a type. An object is what you actually use in a program.
- > A **method (member function)** is a function contained within the class. You will find the functions used within a class often referred to as methods in programming literature

C++ Terminology

- > A **message** is the same thing as a function call.
 - In object oriented programming, we send messages instead of calling functions.
 - For the time being, you can think of them as identical.
 - Later we will see that they are in fact slightly different.

Conclusion

- > Until this slide we have discovered some features of the object-oriented programming and the C++.
- > Our programs consist of object as the real world do.
- > Classes are living (active) data types which are used to define objects. We can send messages (orders) to objects to enable them to do something.
- > Classes include both data and the functions involved with these data (**encapsulation**).
- > As the result:
 - Software objects are similar to the real world objects,
 - Programs are easy to read and understand,
 - It is easy to find errors,
 - It supports modularity and teamwork.

Defining Methods as inline Functions

- > In the previous example, only the prototypes of the member functions are written in the class declaration.
- > The bodies of the methods are defined outside the class.
- > It is also possible to write bodies of methods in the class.
- > Such methods are defined as inline functions.

Defining Methods as inline Functions

- > For example the **isZero** method of the **Point** class can be defined as an **inline** function as follows:

```
class Point{
    int x,y;
public:
    void move(int, int);
    void print();
    bool isZero() {
        return (x == 0) && (y == 0);
    }
};
```

Defining Dynamic Objects

- > Classes can be used to define variables like built-in data types (int, float, char etc.) of the compiler.
- > For example it is possible to define pointers to objects. In the example below two pointers to objects of type Point are defined.

```
int main() {
    Point *ptr1 = new Point;
    Point *ptr2 = new Point;
    ptr1->move(50, 50);
    ptr1->print();
    ptr2->move(100, 150);
    if( ptr2->isZero() )
        cout << " Object pointed by ptr2 is on zero." << endl;
    else
        cout << " Object pointed by ptr2 is NOT on zero." << endl;
    delete ptr1;
    delete ptr2;
    return 0;
}
```

Defining Array of Objects

- > We may define static and dynamic arrays of objects. In the example below we see a static array with ten elements of type Point.
- > We will see later how to define dynamic arrays of objects.

```
int main() {
    Point array[10];
    array[0].move(15, 40);
    array[1].move(75, 35);
    :
    for (int i = 0; i < 10; i++)
        array[i].print();
    return 0;
}
```

Controlling Access to Members

- > We can divide programmers into two groups: class creators (those who create new data types) and client programmers (the class consumers who use the data types in their applications).
- > The goal of the class creator is to build a class that includes all necessary properties and abilities. The class should expose only what's necessary to the client programmer and keeps everything else hidden.
- > The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development.

Controlling Access to Members

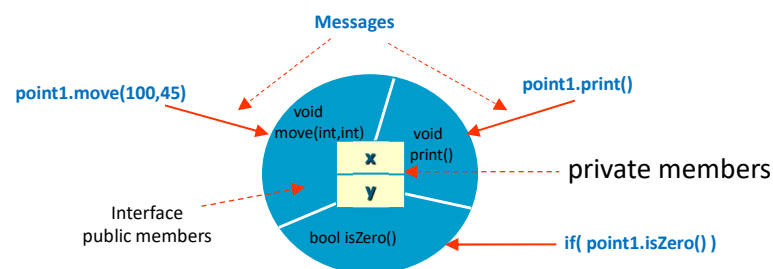
- > The first reason for access control is to keep client programmers' hands off portions they shouldn't touch.
- > The hidden parts are only necessary for the internal machinations of the data type but not part of the interface that users need in order to solve their particular problems.
- > The second reason for access control is that, if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact to anyone else.
- > This protection also prevents accidentally changes of states of objects.

Controlling Access to Members

- > The labels **public:** , **private:** (and protected: as we will see later) are used to control access to a class' data members and functions.
- > private class members can be accessed only by members of that class.
- > public members may be accessed by any function in the program.
- > The default access mode for classes is private:
- > After each label, the mode that was invoked by that label applies until the next label or until the end of class declaration.

Controlling Access to Members

- > The primary purpose of public members is to present to the class's clients a view of the services the class provides. This set of services forms the public interface of the class.
- > The private members are not accessible to the clients of a class. They form the implementation of the class.



Example

- > We modify the move function of the class Point. Clients of this class can not move a point outside a window with a size of 500x300.

```
class Point {
    int x, y;
public:
    bool move(int, int);    // A function to move the points
    void print(); // to print the coordinates on the screen
    bool isZero(); // is the point on the zero point (0,0)
};

bool Point::move(int new_x, int new_y) {
    if( new_x > 0 && new_x < 500 && new_y > 0 && new_y < 300) {
        x = new_x;
        y = new_y;
        return true;
    }
    return false;
}
```

Example

- > The new move function returns a boolean value to inform the client programmer whether the input values are accepted or not:

```
int main() {
    Point p1;
    int x,y;
    cin >> x >> y;
    if ( p1.move(x,y) )
        p1.print();
    else
        cout << "\nInput values are not accepted";
}
```

- > It is not possible to assign a value to x or y directly outside the class.
 - **p1.x** = 42; //**ERROR!** x is private

struct Keyword in C++

- > **class** and **struct** keywords have very similar meaning in the C++.
- > They both are used to build object models.
- > The only difference is their default access mode
 - The default access mode for class is **private**
 - The default access mode for struct is **public**

Friend Functions and Friend Classes

- > A function or an entire class may be declared to be a friend of another class.
- > A friend of a class has the right to access all members (private, protected or public) of the class

```
class A{  
    friend class B;  
    private:  
        int i;  
        float f;  
    public:  
        void fonk1(char *c);  
};
```

```
class B{  
    int j;  
    public:  
        void fonk2(A &s) {  
            cout << s.i;  
        }  
};
```

*In this example, A is not a friend of B.
A can not access private members of B.*

Friend Functions and Friend Classes

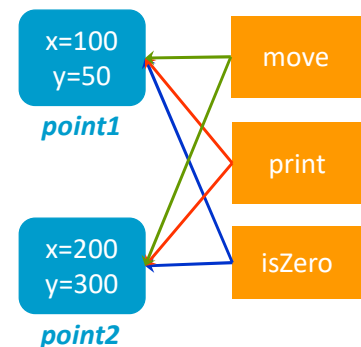
- > A friend function has the right to access all members (private, protected or public) of the class.

```
class Point {  
    friend void zero(Point &);  
    int x,y;  
    public:  
        bool move(int, int);  
        void print();  
        bool isZero();  
};
```

```
// Assigns zero to all coordinates  
void zero(Point &p) {  
    p.x = 0;  
    p.y = 0;  
}
```

this Pointer

- > Each object has its own data space in the memory of the computer.
- > When an object is defined, memory is allocated only for its data members.
- > The code of member functions are created only once.
- > Each object of the same class uses the same function code.
- > How does C++ ensure that the proper object is referenced?
- > C++ compiler maintains a pointer, called the this pointer.



this Pointer

- > A C++ compiler defines an object pointer **this**.
- > When a member function is called, this pointer contains the address of the object, for which the function is invoked.
- > So member functions can access the data members using the pointer **this**.
- > Programmers also can use this pointer in their programs.

this Pointer

- > Example: We add a new function to Point class: **far_away**.
- > This function will return the address of the object that has the largest distance from (0,0)

```
Point* Point::far_away(Point &p) {  
    unsigned long x1 = x*x;  
    unsigned long y1 = y*y;  
    unsigned long x2 = p.x * p.x;  
    unsigned long y2 = p.y * p.y;  
    if ( (x1+y1) > (x2+y2) ) return this;  
    return &p;  
}
```

this Pointer

- > **this** pointer can also be used in the methods if a parameter of the method has the same name as one of the members of the class.

```
class Point{  
    int x,y;  
public:  
    bool move(int, int);  
    :  
};
```

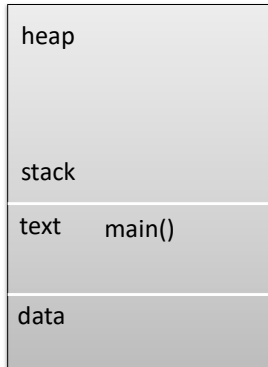
```
bool Point::move(int x, int y) {  
    if( x > 0 && x < 500  
        && y > 0 && y < 300) {  
        this->x = x;  
        this->y = y;  
        return true;  
    }  
    return false;  
}
```

SUMMARY

CLASSES AND OBJECTS

Summary

Process Model



Point o; o ^S
o.move(1,1);



Summary

Process Model

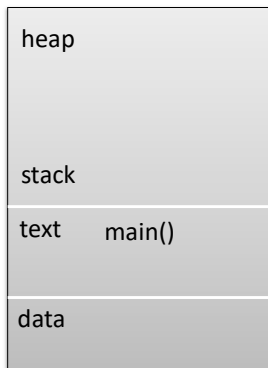


Point *p; p ^S
 ● → ?

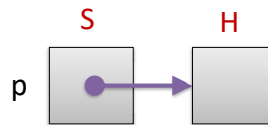


Summary

Process Model



Point *p;



```
p= new Point();
```

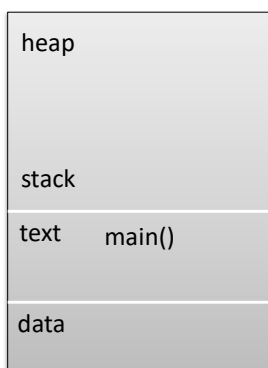
```
p->move(1,1);
```

```
(*p).move(1,1);
```

```
p[0].move(1,1);
```

Summary

Process Model

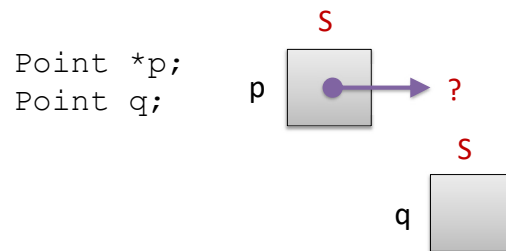
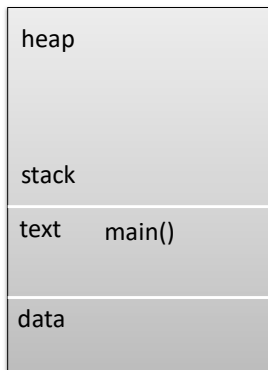


```
Point *p;
```

```
Point q;
```

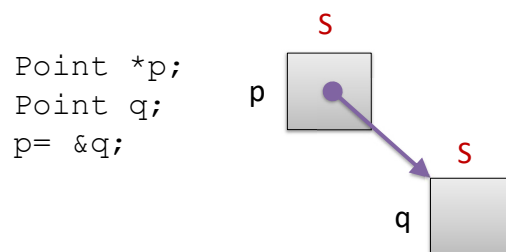
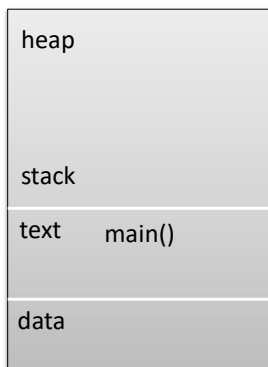
Summary

Process Model



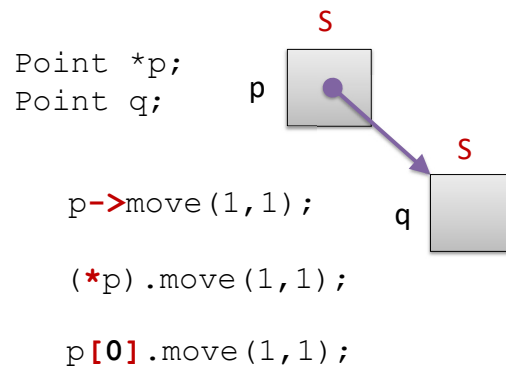
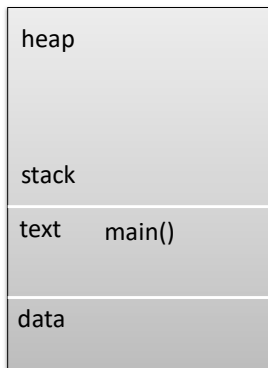
Summary

Process Model



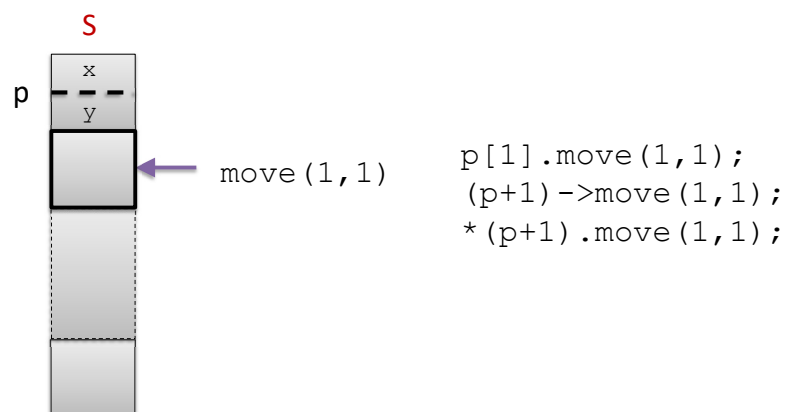
Summary

Process Model



Summary

Point p[10];



Summary

